

# StreamNF: Performance and Correctness for Stateful Chained NFs

Junaid Khalid, Aditya Akella  
University of Wisconsin-Madison

**Abstract:** Network functions virtualization (NFV) – deploying network functions in software on commodity machines – allows operators to employ rich chains of NFs to realize custom performance, security, and compliance policies, and ensure high performance by dynamically adding instances and/or failing over. Because NFs are stateful, it is important to carefully manage their state, especially during such dynamic actions. Crucially, state management must: (1) offer good performance to match the needs of modern networks; (2) ensure NF chain-wide properties; and (3) not require the operator to manage low-level state management details. We present StreamNF, an NFV framework that satisfies the above requirements. To do so, StreamNF leverages a external state store with novel caching strategies and offloading of state operations, and chain-level logical packet clocks and packet logging/replay. Extensive evaluation of a StreamNF prototype built atop Apache Storm shows that the significant benefits of StreamNF in terms of state management performance and chain-wide properties come at a modest per-packet latency cost.

## 1 Introduction

NFV allows operators to overcome failure and performance issues of software NFs by quickly spinning up additional/backup instances of an NF and dynamically redistributing processing to them (e.g., using SDN). Furthermore, NFs can be chained together [2, 13] to implement custom policies, and frameworks exist for specifying such chains [19, 20].

Because NFs are *stateful* it is important to ensure that updates to NF-internal state satisfy certain key properties, especially as traffic is being reassigned across instances (e.g., during scale up) [14, 27]. In the context of NFV chains deployed in modern networks, two additional requirements arise: (a) the state management mechanisms that ensure these properties must offer high performance, and (b) they must work across the entire chain, and not at the level of individual NFs in a chain.

We present StreamNF, a new NFV framework. It exposes a simple API for operators to specify and customize NFV chains. StreamNF’s runtime hides the complexity of supporting high chain performance and fault tolerance while ensuring consistent chain-wide actions.

Many NFV frameworks exist today, but they suffer from key drawbacks. First, their design choices impose **high performance overhead** on state management. For instance, both OpenNF [14] and FTMB [27] rely on maintaining a separate, up-to-date copy of NF state to ensure fault tolerance, which adds both storage and processing overhead. Second, existing frameworks manage (some types of) state updates across instances of a *single NF*. E.g., FTMB and Pico Replication [27, 23] maintain up-to-date copies of state shared across a given NF’s instances; Split/Merge [24] keeps state up-to-date as it is being moved across an NF’s instances. These approaches fundamentally **cannot support chain-level properties**, e.g., tracking the order in which updates were made to an arbitrary NF’s state relative to the order of chain input arrival. Third, they expose **low-level details**. For example, OpenNF requires an operator to be aware of which instances are processing specific flows and choosing the algorithm to reallocate state across instances.

To improve the performance of state management, StreamNF combines three main ideas. (1) StreamNF externalizes all NF state to an in-memory store (inspired by [15]). This ensures that NF state is available beyond instance failure. (2) StreamNF uses state scope-awareness to partition traffic to instances such that an instance mostly updates state objects that other instances cannot (or rarely) update, minimizing cross-instance state access [19, 11]. Crucially, StreamNF uses state scope to implement effective caching strategies. (3) Because most NFs today perform a simple set of state update operations, StreamNF offloads all such operations to the state store; the store performs the operation and updates state in the background on behalf of an NF instance. This innovation speeds up cross-instance state updates – all shared state coordination is handled by the state store which serializes the operations issued by multiple instances. It also helps separate NF processing from the overhead of updating externalized state.

The state store provides APIs that simplify an NF’s access to another NF’s state (e.g., by offering read-upon-update). This is key to interesting cross-NF analyses, which are not well supported today.

To ensure chain-wide properties StreamNF introduces special NFs at chain heads (hidden from the admin)

that: (1) apply a unique logical clock to every incoming packet, and (2) log packets whose processing is still ongoing at some instance in the chain. The logged packets are replayed to bring up to speed backup instances (these could be failover or clone instances). StreamNF’s chain-level view of clocks and logging is more powerful than NF-centric approaches [27]. Logical clocks ensure NFs upstream from the backup instance don’t make duplicate state updates during replay. Logical clocks can be used to check/ensure chain-wide ordering properties.

StreamNF exposes a natural DAG-based API to specify NF chains. Operators can customize the DAG, e.g., by providing custom NF implementations. The StreamNF execution framework handles elastic chain scale up/down, fast fail-over, and cloning/restarting slow NFs, while handling relevant state management issues that are required for chain-wide guarantees.

We have prototyped StreamNF atop the Apache Storm stream processing system [1]. We use Redis [6] for the state store. We have implemented five different NFs to run atop StreamNF. We evaluated this prototype via extensive experiments that use campus-to-EC2 packet traces. Key findings from our evaluation are: State externalization adds per packet latency overhead, but StreamNF’s optimization (e.g., caching and ack-offloading §7) reduce it to  $\sim 2 - 3\mu\text{s}$  per packet. StreamNF’s state move operations finish 95% faster than OpenNF’s, and its fault tolerance support offers 4X better per packet latency than FTMB. Finally, StreamNF’s support for chain-wide guarantees eliminates the false positives/negatives seen when using certain security NFs with existing NFV frameworks.

## 2 Motivation

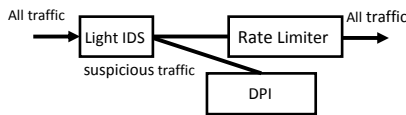


Figure 1: *Example NFV Chain.* All traffic must pass through a light-weight intrusion detection system (IDS). A rate-limiter throttles traffic marked as suspicious by the IDS; other traffic is unaffected. A copy of suspicious traffic is sent to an off-path deep packet inspection (DPI) engine for further analysis.

NFV allows network operators (henceforth “admins” or “users”) to realize complex NF policies by connecting multiple NFs in chains [4, 13, 9, 5]. An example chain is shown in Figure 1. Since the NFs are typically run in software, they cannot handle arbitrary load – high input traffic volumes can cause unreasonably high latencies. Also, they are susceptible both to *hard* failures, e.g., due to crashes, as well as *soft* failures, e.g., compute or

network contention degrading the throughput of an NF. Hence, each stage of processing in a chain needs the ability to scale up to handle load, and add backup instances in the case of hard/soft failures.

A central issue is ensuring that the NFs as a whole take actions that the operator intended. Given an input packet sequence, the output produced by an NF chain (packets and/or events generated) should match that taken by an equivalent NF chain with a single instance per NF and NF-NF links both with unbounded capacity. Crucially, this must hold even as traffic is being dynamically re-allocated across instances as scale up/down or failover actions are taken. We call this *chain output equivalence*.

An ideal NFV framework must support: (I) chain equivalence, (II) APIs for specifying chains and deploying rich NFs, and (III) high performance.

### 2.1 Required Functionality

**I. Chain equivalence:** Ensuring chain equivalence is challenging because: (a) NFs are stateful in nature: each NF maintains detailed state objects for individual or groups of flows; the state may be updated by each incoming packet and the current value of the state may be used in determining the action to take on a packet. (b) NF chaining might require that the action taken by an NF be dependent on the actions taken by upstream instances.

Ensuring arbitrary chain equivalence is difficult. For all practical purposes though, it boils down to the following requirements for *cross-instance* (R1–R3) and *cross-NF* (R4, R5) state management:

- (R1) *State availability:* When an NF instance fails, all state it has built up internally disappears. A backup instance can take over processing as long as it is initialized with the exact state of the failed instance prior to failure [14]. This requires NF-internal state to be made highly available.
- (R2) *Cross-instance state transfers:* In some cases, live traffic must be reallocated across NF instances because the admin wishes to quickly rebalance load. In such cases, the internal state corresponding to the reallocated traffic (which exists at the old location where the traffic was being processed) must be made available at the traffic’s new location. Updates to state due to packets that were in transit to the old location prior to redistribution must be reflected in the state’s new location. When either is violated, an NF can take incorrect action [14].
- (R3) *Cross-instance state sharing:* Depending on the nature of an NFs’ state, it may not be possible to completely avoid sharing it across instances. Updates to such state at any instance may need to be kept consistent with other instances that share the state.

- (R4) *Chain-wide ordering*: Because of variations in the processing speeds of different instances in a chain, it is possible that traffic belonging to different subsets is delayed by arbitrary amounts across a chain. An NF that relies on knowing the exact order in which traffic entered the chain may take different decisions for the same input traffic based on intervening NF’s processing rates.
- (R5) *Duplicate suppression*: The impact of slow-running instances (a “soft” failure), or stragglers, can be mitigated by deploying clones. The original instance and the clone process the same input simultaneously, and the one making faster progress is retained. In such cases, the original and clone instances will both generate duplicate alarms and output traffic. To ensure downstream NFs’ actions are not impacted these duplicates must be suppressed.

**II. Chains with rich NFs:** The NFV framework should place no restrictions on the nature of NFs used or the policies specified in NFV chains (R6).

Many interesting policies require some form of *cross-NF analysis* (R7). Consider a logical two-NF chain. The first is an on-path firewall that: (a) drops all traffic from “blocked” hosts or prefixes, and (b) logs for each host the number of (un)successful connection attempts to the host. The second is an off-path analyzer that parses content signatures to determine if hosts are to be blocked, but its accuracy can be improved by comparing the ratio of successful to unsuccessful attempts for a host with the average ratios for all other hosts. This can be supported by allowing the analyzer to access the firewall’s state.

**III. Performance:** The mechanisms that support requirements R1–R7 need to be *high-performance* (R8) to support production traffic.

## 2.2 Related work, and Drawbacks

A variety of NFV framework exists today. However, *none of them support cross-NF requirements R4 and R5*. A handful support cross-NF analysis (R7) but in a rather poor fashion. Many of the frameworks focus on handling a subset of the cross-instance requirements R1–R3, but the mechanisms used impose a high performance overhead (R8). Finally, a few frameworks offer high level APIs to specify chains (R6), but they don’t support state management (R1–R5).

*Cross-instance requirements*: FTMB [27], StatelessNF [15], and Pico Replication [23] focus on state availability; OpenNF [14] and Split/Merge [24] optimize cross-instance state transfers. Unfortunately, “composing” these techniques to combine their benefits is difficult as they are incompatible with each other.

*High-level APIs*: Many existing approaches don’t offer high-level APIs to network operators. They often deal with a single type of NF (e.g., FTMB, Pico Replication and Split/Merge). OpenNF can support chains, however, an operator must still be aware of which instances are processing specific flows and how to reallocate state across instances. E2 [19] and PGA [20] offer abstractions for specifying chains, but provide no support for state management (R1–R5).

*Cross-NF analysis*: Existing frameworks also don’t support cross-NF analysis well enough. Consider the example above. FlowTags [13] and E2 [19] enable the firewall and analyzer to exchange limited contextual information alongside the traffic flowing between them. But they impose two constraints: (a) Each NF can only operate on the narrow contextual information that the other NF chooses to expose (e.g., whether a firewall rule matched), as opposed to taking an informed action by reading relevant runtime state of the other NF when needed; also, bidirectional traffic flow is needed to exchange context information. (b) Traffic may be unnecessarily replicated/forwarded. The analyzer only needs to know the number of connection attempts across all hosts; having access to the entire input traffic is unnecessary.

*State management performance*: Existing frameworks’ state management performance is poor. For instance, OpenNF requires a central controller to periodically merge shared state across a collection of  $n$  NF instances, requiring periodic copying across  $n^2$  NF instance-pairs. OpenNF’s state transfer operations impose high per packet latency because they require a controller to orchestrate state movement across instances in concert with network forwarding updates.

Other frameworks not listed [10, 7, 26] also fall short for the same reasons as above.

## 3 API

StreamNF is a novel NFV framework that meets the aforementioned requirements. StreamNF is inspired by similar frameworks in big data analytics systems [12, 28, 25]—similar to such frameworks, StreamNF provides intrinsic support for elastic scale up/down, fault tolerance, and straggler mitigation, while hiding from the application programmer the underlying details. We describe StreamNF’s high-level API for admins first.

### 3.1 Logical Processing DAGs

The structure of underlying NF processing required to support policies such as Figure 1 maps naturally to directed acyclic graphs (DAGs) of computational elements. Therefore, we provide a DAG API, building on and extending prior work [19, 10, 18], to allow users to express

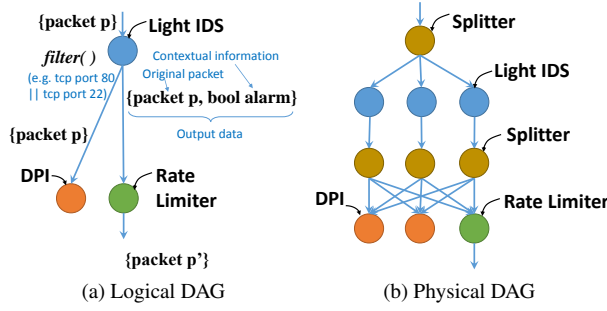


Figure 2: Logical, physical DAGs for the example in Figure 1

```

1 logicalDAG dag = new logicalDAG();
2 vertex vIDS = new vertex(IDS.executor);
3 vertex vDPI = new vertex(DPI.executor);
4 vDPI.properties(hw.packetParser);
5 vertex vRatelimiter = new vertex(ratelimiter.
    executor);
6
7 edge e1 = new edge(vIDS.flowSpace());
8 edge e2 = new edge(vDPI.customSplitter, filter());
9 edge e3 = new edge(vRatelimiter.flowSpace());
10 edge e4 = new edge();
11
12 dag.addVertex(vIDS).addVertex(vDPI)
13   .addVertex(vRatelimiter);
14
15 dag.addEdge(dataSrc.out_port, vIDS.in_port, e1).
    addEdge(vIDS.out_port1, vDPI.in_port, e2).
    addEdge(vIDS.out_port2, vRatelimiter.in_port,
    e3).addEdge(vRatelimiter.out_port, dataSink.
    in_port, e4);

```

Figure 3: DAG specification for the example in Figure 2a.

lower-level details of their chains’ NF processing, specifically, the NFs’ input/output traffic and behavior, dependencies, NF processing logic, and requirements.

Figure 3 shows the **logical processing DAG** corresponding to the policy in Figure 1 expressed using our API. Each vertex is a processing unit; an NF can be represented as a single vertex or a DAG of vertices. In our example, each vertex maps to a different NF.

Each vertex may: (a) analyze incoming packets; (b) transform packets; (c) generate contextual output (e.g., raise specific events or alarms), and (d) update state objects (more below). Edges represent data flow, and edge annotations reflect aspects of the data flowing between vertices (e.g., the specific set of flows).

Each **vertex** is a logical processing unit; vertex declarations are shown in lines 2, 3 & 5 in Figure 3. The vertex definition consists of user supplied NF code, input and output classes, configuration, and state objects.

User-supplied code (e.g., “IDS.executor”) defines the internal packet processing logic of the vertex. We expect the NF author (potentially distinct from the admin)

to provide this. A vertex can emit one or more types of outputs for different downstream vertices. These can be packets or contextual output<sup>1</sup>; contextual output, represented using JSON, can be simple boolean values (e.g., suspicious vs. not), or it can be arbitrarily rich (e.g., the set of signatures associated with a suspicious host). E.g., as shown in Figure 2, the IDS can emit packets for a downstream off-path DPI for further analysis, and alarms for suspicious flows along with packets for a downstream rate limiter. Likewise, a vertex can accept one or more types of input for processing. To accommodate this, in our DAG, each vertex can have one or more input/output ports (similar to Click [18]) as shown in line 15 of Figure 3. Each input (output) port is associated with a class. A pair of vertices can communicate with each other via a directed edge between an output port on the upstream and input port of the downstream only if the classes of the two ports match.

A vertex configuration captures vertex requirements. These span: (a) resource requirements (e.g., in terms of expected minimum CPU, memory and network bandwidth needed), and (b) the ability to leverage special hardware capabilities where available (e.g., hardware encryption engine or NIC-supported packet parsing) as shown in line 4 in Figure 3.

Finally, vertex definitions include state objects. These maintain the corresponding NF’s runtime internal state. For each of these state objects, the NF author provides the *flowspace* of the object, which is the set of attributes of incoming traffic (packet header fields) the state is keyed on. An object whose flow-space is the connection 5 tuple<sup>2</sup> is called a “per-flow” state object. The 5-tuple is the *finest* granularity flowspace that can be specified.

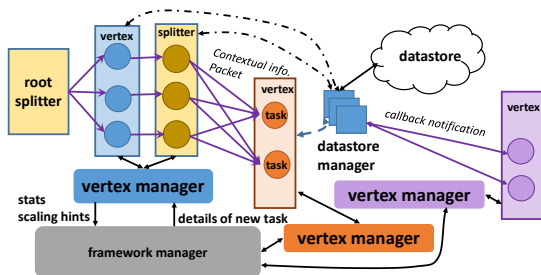
**Edge** represents the flow of data (packets and/or contextual output) from an upstream vertex to a downstream vertex. Edges carry annotations: a traffic filter for an edge (line 8 Figure 3) defines what subset of traffic goes over the edge as per the user’s policy. Contextual output for an edge indicates the type for the information the upstream vertex forwards to the downstream.

By default a vertex partitions traffic based on the flowspace of the state objects of the downstream vertices (lines 7 & 9 in Figure 3). The goal is to minimize cross-instance state sharing. The StreamNF framework handles and hides from the user the complexity of partitioning at an upstream vertex and gathering traffic at a downstream vertex (§4.2).

For a pair of NFs to coordinate, the admin must ensure that the output class of the upstream vertex matches the

<sup>1</sup>We will use ‘output’ to refer to both packet and contextual outputs.

<sup>2</sup>(srcIP, srcPrt, dstIP, dstPrt, proto)



input class of the downstream vertex. The semantics of the input and output classes that a particular vertex can consume or generate are specified by the NF vendor.

### 3.2 Physical Processing DAGs

StreamNF compiles the above logical DAG into a corresponding physical DAG. Each logical vertex is mapped to a collection of one or more *tasks* in the physical DAG; these are instances of the logical vertex (we use the terms *task* and *instance* interchangeably where the context is clear). For example, the IDS in Figure 2a is mapped by StreamNF to three vertices in the physical DAG as shown in Figure 2b. Each task processes in parallel a partition of the entire input to the logical vertex. The extent of parallelism is determined at run time, e.g., based on input load (§4.1).

## 4 StreamNF Building Blocks

We present four key building blocks of StreamNF – vertex manager, scope-aware splitter, communication via message queues, and a distributed state store (Figure 4). StreamNF’s dynamic vertex and state management actions are both informed (when to act) and aided (how to act for chain equivalence) by these.

## 4.1 Vertex Manager

StreamNF provides intrinsic support for elastic scaling, fault tolerance and straggler mitigation (§5). Users can customize the logic used for each of these three aspects by providing modules for them. These modules work on the global view of the DAG by taking hints from *vertex managers* as input. The vertex manager, shown in Figure 4, inspired by Tez [25], is responsible for collecting statistics from each vertex’s instances to provide hints to the framework (e.g., for scale up/down or straggler mitigation; §4.3 and §5). These hints are customizable through user defined modules.

## 4.2 Scope-aware partitioning and splitters

StreamNF performs scope-aware partitioning of data flowing across two sets of NF instances corresponding to

neighboring NF vertices in the logical DAG. This allows StreamNF to minimize state sharing across instances, and hence cross-instance coordination.

To support this, StreamNF leverages the flowspace information associated with each state object as follows.<sup>3</sup> The function `flowspace()` in line 7 and 9 in Figure 3 returns a list of flowspaces (the set of packet header fields) which are used to key into the objects that store the state for the particular NF. The list is ordered from the most fine grained flowspace to the most coarse grained one. Consider the example of an IDS vertex with two state objects: one corresponding to records of whether a connection is successful or not; and another corresponding to number of connections per host. The flowspace for the former is defined by the 5-tuple (*src IP*, *dst IP*, *src port*, *dst port*, *protocol*); the flowspace for the latter object is *src IP*. StreamNF first attempts a partitioning of the traffic at an instance immediately upstream (from the IDS instance in our example) based on the most coarse-grained flowspace (*src IP* in our example); such splitting results in no state sharing at the downstream NF instances. However, it may not be appropriate because it results in an uneven load split across instances; the framework gathers this information via the downstream vertex manager’s hints. In such a situation, the framework considers progressively finer grained flowspaces for partitioning at the splitter. In the worst case, splitting happens on the finest-grained flowspace (e.g., the connection 5-tuple in our example).<sup>4</sup>

The flowspace to partition on is provided as input to *splitters*. A splitter is a special task inserted after every NF instance in a physical DAG as shown in Figure 2b. It is responsible for partitioning the output traffic of the task to instances immediately downstream.

The root vertex of a physical DAG is also a splitter, but it additionally parses the packet headers and annotates the packet with metadata to avoid the overhead of reparsing the headers on every vertex. It also computes logical packet timestamps and performs packet logging §5.

Note that partitioning applies to both packet and contextual information output by a task.

### 4.3 Communication

Communication between vertices is asynchronous and non-blocking. Each vertex can generate one or more outputs for a particular input, which are received by the StreamNF framework. StreamNF is responsible for dis-

<sup>3</sup>Other work has used program analysis tools to infer flowspaces and perform scope-based partitioning [16, 11], but our job is easier as the relevant input is available directly.

<sup>4</sup>Which inevitably means state pertaining to number of connections per host is shared across instances.

Operation	Description
Increment/ decrement a value	Increment or decrement the value stored at key by the given value.
Push/pop a value to/from list	Push or pop the value in/from the list stored at the given key. If the list does not exist, create a list.
Compare and update	Update the value, if the condition is true.

Table 1: Basic operations offloaded to datastore manager

tributing and routing the output to downstream tasks. The framework stores all the outputs received from the upstream tasks in a queue at the downstream task, and the downstream tasks polls the queue for input.

This approach offers three advantages: (a) upstream tasks can produce output independent of the instantaneous consumption rate of the downstream task (this is a well understood benefit of message queues [8]), (b) the framework can operate on the queue contents (e.g., delete specific messages before they are processed by the downstream vertex), which is useful in realize chain-wide properties (e.g., when stragglers are cloned), (c) queue attributes (e.g., persistent high occupancy at some queues) can be used by the relevant vertex manager as a hint to identify bottlenecks (either a straggler instance or high aggregate load).

#### 4.4 Storage

In most existing NFV frameworks today, NF instances maintain critical state internally. As mentioned earlier, state management is rigid (in that sharing and coordination are not well supported) and can offer low performance.

StreamNF offers flexible high performance state management by *externalizing* the internal state of NFs, and storing it in an external distributed key-value datastore. Crucially, NF instances update their internal state via issuing state *operations* directly on the datastore. As we argue below, this eliminates/hides the complexity of cross-instance state reads/writes while also helping improve performance of state management actions, and ultimately of NF processing.

All operations are managed by the **Datastore Manager** (Figure 4). It’s design is based on two key insights.

First, most NFs today perform simple operations on their internal state. Table 1 shows common examples. We offload these common state operations to the datastore manager: instead of reading from, operating on, and then writing state back to the datastore, a task can instruct the datastore manager to perform the specific operation listed in Table 1 on the state on its behalf.<sup>5</sup> We discuss

<sup>5</sup>Developers can load custom operations.

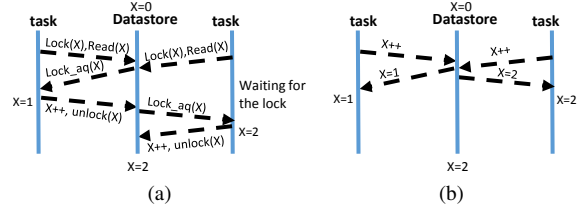


Figure 5: State updates: (a) locking, (b) StreamNF

datastore manager scalability in §7.

The key benefit is that NF instances do not have to contend for shared state. The datastore manager serializes the operations requested by different instances for the same object (Figure 5b). This vastly reduces the overhead compared to, e.g., a naive approach, in which an NF task first acquires a lock on the state, updates it, and releases the lock (Figure 5a).

Our second insight is that in many cases, upon receiving a packet, an NF updates (writes to) a state object, but does not use (read) the updated value. An example is a packet counter, which each task updates for every input packet, but the updated value is only read infrequently. Thus, we offer *non-blocking* semantics for updates: the datastore manager immediately sends the requesting task an ACK for the operation, and applies the update later in the background. However, if a task wishes to read the value, the datastore manager applies all the outstanding updates before serving the read request.

**State metadata:** The storage manager’s client-side library at a task appends metadata to the key of the state the task stores. The metadata consist of DAG\_ID, vertex\_ID, and task\_ID. These IDs are immutable and are assigned by the framework. DAG\_ID is used to distinguish between the state objects of different DAGs of the same user or multiple users.

The key for a per flow state object is:  
 $DAG\_ID + vertex\_ID + task\_ID + object\_key$   
 where object\_key is the ID assigned to the state object at the time of DAG initialization. This ensures that only the task to which the flow is assigned can update the corresponding state object.

The key for all other objects in the datastore, e.g., *packet.count* is:  $DAG\_ID + vertex\_ID + object\_key$   
 All the tasks of a logical vertex can update such objects. When two logical vertices use the same key to store their state, vertex\_ID prevents any conflicts.

**Caching:** To further improve the performance and reduce state update latency, the datastore’s client-side library caches state objects. Based on the keys assigned above, and because StreamNF ensures that flows that can update such objects are processed by a single instance at any time, per-flow state objects do not have



cross-instance consistency requirements; thus they can be safely cached at the instance.

Any other state object is only cached when tasks rarely update it – a developer can identify an object as read-heavy; additionally scope-aware partitioning may result in instance-specific updates to a non-perflow object. For such objects, the client-side library registers a callback with the datastore manager, which is invoked whenever some other task pushes an update to the object. Upon receiving the callback, the client-side library updates the locally cached value.

Additionally, for such objects, the datastore can allow them to be cached as long as no other task is accessing them. When such an access occurs, the datastore manager notifies the client-side library to stop caching that object and flush any cached state; in fact the library only needs to flush the corresponding state *operations*, which imposes far less overhead than flushing state.

**Callbacks:** StreamNF allows vertices to register callbacks with the datastore manager for timer and state update events. The latter can be used by a task to know about updates to cached cross-flow state (as exemplified above) or for cross-instance coordination (R7; §2).

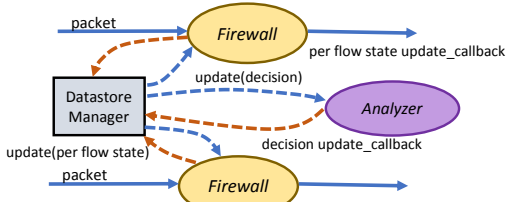


Figure 6: Cross-NF analysis

Recall the example of the firewall and the analyzer from §2.1. Callbacks vastly simplify and improve the performance of the cross-NF coordination needed in this case (Figure 6): The firewall updates simple per-host state (number of (un)successful connections for the host) at the store, and the analyzer has simple per-host state (on whether to block it or not). The analyzer registers a callback with the datastore manager to be notified whenever any host’s state value is updated. Likewise, the firewall registers a callback with the datastore manager to be notified whenever the *analyzer* updates the decision for a host to be blocked or not. This prevents all traffic from being mirrored to the analyzer, and the needed communication between the analyzer and the firewall happens in a simple indirect fashion via the datastore.

## 5 Runtime management

StreamNF includes a *framework manager* that supports high performance for an admin’s NFV chain while en-

suring chain equivalence. To achieve this, the framework manager leverages the building blocks from the previous section in order to support: (a) quickly reacting to hard failures by bringing up new instances, (b) reacting to soft failures by killing or duplicating slow running stragglers, and (c) quickly scaling up the vertices in a DAG to meet demand (and scaling down for cost-efficiency).

Two key mechanisms – creation/use of logical timestamps, and packet logging – aid the framework manager in ensuring chain equivalence.

**Logical clocks and logging:** The root splitter in the physical DAG associates with every new input packet a *logical clock* value (passed as metadata along with the packet in our current implementation) that uniquely identifies the packet. It is incremented per packet. The current clock value is stored by the splitter in the datastore.

In addition, the root splitter logs all input packets along with their logical timestamps in the datastore. The splitter also tracks to which immediate downstream task a packet was forwarded.

The logical clock added by the splitter at the root vertex is “consumed” at the most immediate downstream *connection terminating* NF (acting as a TCP endpoint, e.g., an application gateway). That is, the instance of such an NF informs the framework that it is done processing a packet (along with updating state and generating any relevant output), at which point the framework sends a “delete” request with the logical clock value of the packet to the upstream root splitter.

Thus, at any point the root splitter stores all packets that are being actively processed by one or more instances between itself and a downstream connection terminating instance. We assume that NFs that are not connection-terminating don’t modify the logical clock passed with a packet.<sup>6</sup>

The splitter immediately downstream from a connection terminating NF also acts as a root splitter: adds its own logical clock and logs the packets it outputs.

### 5.1 Fault tolerance

When a task fails, the framework manager deploys a new instance to take over the failed task’s processing. Since we externalize the state, the failed task’s state continues to be available; the manager associates the new task’s ID with this state. However, the new task is not yet ready to take over processing: we must account for packets that were in-transit to the failed task when failure occurred. State updates that such packets would have made had

<sup>6</sup>NF authors can ensure this. Alternately, or as a fail-safe for this assumption, the framework can match incoming and outgoing packets by computing a Rabin Fingerprint [21] over the contents [22] and ensure matching packets have the same logical clock.

failure not occurred would need to be captured in order to ensure correct NF processing.

A simple solution is to replay all logged packets from the root splitter vertex. The splitter continues to forward the new incoming packets alongside replayed ones. To indicate the end of replayed traffic, the root splitter marks the last replayed packet (this is the last logged packet at the time the root started replaying). The framework manager buffers the new incoming traffic at the new task, until it receives the replayed packet marked as “last”; buffered packets are then released for processing.

However, the above simple approach can lead to incorrect NF actions due to two reasons: (1) The failed task may have updated only a subset of state objects prior to crashing. For example, the IDS in Figure 7a updates both the total packet count, and the number of active connections per host. An IDS instance may crash after updating the former but without updating the latter. In such cases, processing a replayed packet can *incorrectly* update the same state object (total packet count) multiple times. (2) NFs upstream from the failed instance would have already processed some of the in-transit packets. In such cases, replaying may cause them to incorrectly process such packets again (e.g., an IDS may raise false alarms).

To address the first issue, the datastore manager keeps along with each state update request issued by a task, the logical clock value of corresponding packet. During replay<sup>7</sup>, when a task sends an update for a state object, the datastore manager checks if an update corresponding to the logical timestamp of the replayed packet has already been applied; if so, the datastore manager emulates the execution of the update by returning the value corresponding to the update, as shown in Figure 7c.

To address the second issue, each replayed packet is marked and it carries the ID of the new task where it will be reprocessed. Such packets need limited special handling<sup>8</sup>: the intervening tasks recognize that this is not a suspicious duplicate packet; if necessary, the tasks read the datastore for state values corresponding to the replayed packet; they then make any needed modifications to packet header fields and produce relevant contextual information (both of these could be functions of the state at the time the packet was processed); the tasks can issue updates to state, too, but in such cases the datastore emulates updates as before. The ID is cleared once the packet is processed at the new task.

Failures of connection terminating NFs’ instances

<sup>7</sup>The manager informs the immediate upstream root splitter to replay all logged packets whose logical clock value is greater than the minimum of the logical clock values of the packets that made the last updates at the failed task state objects.

<sup>8</sup>We assume that the NF author implements this minimal logic.

need special handling because they consume logical clocks on incoming packets and provide new ones for outgoing packets. Suppose such a task T fails after consuming a packet P and generating another packet P’, but before the framework sends a delete request for P to the upstream root splitter. In such cases, P is replayed, causing T’s new instance to generate P’ but with a *new* logical clock. P’ will be processed anew resulting in state updates at NFs downstream from T, which is incorrect. To address this, our framework suppresses any output at connection terminating NFs that was generated in response to a replayed input for which all state updates were emulated by the datastore manager (meaning that the packet was completely processed earlier).

## 5.2 Straggler mitigation

A straggler is an NF instance whose per-packet processing latency is unusually high, causing the entire NF chain’s performance to suffer. Our framework allows the user to provide custom logic to identify stragglers on a per logical vertex basis – this logic runs in the vertex manager; it can use, e.g., the relative message queue lengths at an NF’s instances to identify stragglers.

To address a straggler, our framework provides support for both killing/restarting it, or cloning it. The former approach is used when resource availability is plentiful and a restarted instance can be launched at a location with the needed compute, memory, and bandwidth. Kill/restart is similar to how we handle task failures.

When resources are scarce, killing/restarting runs the risk of deploying the restarted instance at a sub-optimal (e.g., heavily contended) location, which may further hurt performance. In such cases, StreamNF supports *cloning*. The clone is deployed at the best location possible, and processes the same input as the original in parallel; StreamNF retains the faster one.

The clone is initialized with the straggler’s latest state available in the data store. All packets whose timestamps fall after this state’s last update are replayed to the clone; and, the splitter upstream from the clone/straggler starts replicating input. The clone processes replayed traffic first, and buffers replicated traffic. When replay ends, the clone flushes and processes buffered packets.

At this point, StreamNF must overcome two challenges. First, an NF may produce output on every packet, so simply mirroring the same input to the clone results in duplicate outputs. StreamNF overcomes this challenge at the message queue(s) corresponding to the immediate downstream task(s), by suppressing duplicate outputs associated with the same logical clock (Figure 8).

The second challenge is duplicate updates to state. To address this, as with task failure, the datastore manager



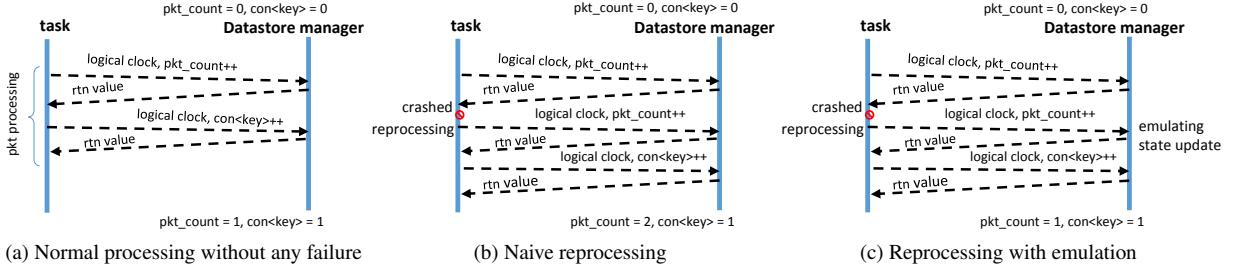
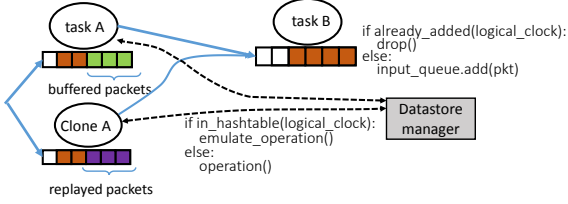


Figure 7: Reprocessing of packet after a failure. Figure a & b illustrate the problem.



keeps track of all the returned values of state update operations associated with the cloned input. When the datastore manager receives a second request (associated with the same logical clock) from the straggler or clone for the state update operation, it returns the value returned when the logical clock was last used in the update.

### 5.3 Elastic scaling

Elastic scale up/down helps ensure that the aggregate processing capacity of the physical DAG matches the input load. A key issue is redistribution of flow processing across instances. A flow may be processed at an “old” task and be reallocated to a “new” task. In such case, StreamNF ensures that the new tasks has the correct state needed to continue processing the flow.

Note that perflow objects in the state store have task IDs in their keys, whereas other objects do not (§4.4). Thus, non-perflow state objects do not need special handling during redistribution. This is in contrast with OpenNF [14] where such shared state must be explicitly copied from/to relevant instances.

However, per-flow state’s handling must be reallocated across instances. A naive approach is that the old task disassociates itself from the state object and the new task associates itself with the state. To disassociate from per flow state, the old task instance just needs to update the metadata of the corresponding state by removing its task\_ID and adding the task\_ID of the new task.

This simple approach works for NFs where the internal state is updated on initial packets (e.g., NAT). But, it does not ensure a safe handover when there are in-transit packets which update the state. Even if the splitter imme-

diately updates the partitioning rules and the traffic starts reaching the new task, there might be packets in-transit or buffered within the old task. If the new task starts processing packets without waiting for the old task to flush the state update associated with in-transit/buffered packets, then the NF may take an incorrect action. This is because the updates due to the in-transit packets may be disallowed by the datastore (as a new task is associated with the state object now) and hence may be lost.

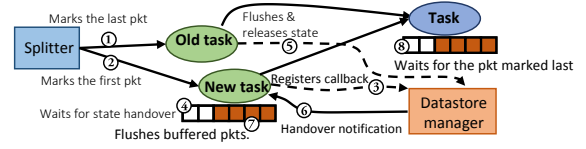


Figure 9 shows the sequence of steps in our framework to address this issue: (1) The splitter marks the metadata associated with the last in-transit packet to inform the old task that the flow has been moved. It should disassociate itself from the per flow state, flush any cached state (operations) associated with the particular flow(s) to the datastore and update the metadata to associate the new task with the per flow state. (2) The splitter also marks the first packet from the flowspace which is being moved to the new task. (3) When the new task receives the marked packet, it tries to access the per flow state from the datastore manager. If the state is still associated with the old task\_ID, it registers a callback with the datastore manager. (4) The new task starts buffering all the packets associated with the flow which is being moved. (5) After processing the packet marked as last, the old task flushes the cached state (operations) and updates the metadata. (6) After the metadata has been updated, the datastore manager notifies the new task about the state handover. (7) The new task associates its ID with the state, and flushes its buffered packets.

Note that the above steps result in ensuring that updates are not lost *and* that they happen in the order in which packets arrived in time (e.g., at the upstream split-

ter). In contrast, OpenNF provides separate algorithms for loss-free and order-preserving updates which impose very different performance overheads; because of these differences, OpenNF leaves it to the admin to choose between the two. However, this is not an easy decision to make as it requires understanding of an NF’s functioning and whether a guarantee matters for an NF’s output equivalence. In contrast, StreamNF absolves the operator from such decisions.

Packets may arrive out of order at a *downstream* task. Here we rely on the message queues: (8) The downstream task does not start processing packets of the moved flow emitted by the new task until the framework enqueues the packet marked as “last” from the old task.

#### 5.4 Additional Comments

**Chain-wide ordering (R4):** Consider a multistep trojan detector [11] which uses cross-flow analysis to identify a trojan. It leverages the fact that a trojan performs the following sequence of steps: (1) opens an SSH connection; (2) downloads an HTML file and a ZIP and EXE file over an FTP connection; (3) generates IRC activity. The order of the steps is relevant; a different order does not necessarily indicate the fingerprint of the trojan.

If this NF were deployed in a chain, then even if a single instance of the NF were used, the ordering of events observed at the instance may be impacted by variations in the processing rates of upstream tasks. Thus, the NF can either incorrectly mark benign traffic as a trojan, or vice versa. When multiple instances of the NF are used, the problem is compounded because it might not be possible to partition the traffic such that all relevant processing happens at one task.

StreamNF simply leverages logical clocks to allow the NF to reason about ordering. The logical clock provides temporal ordering between flows. Consider the more complex case where multiple instances are used. The NF developer can deploy an additional simple analyzer NF that aggregates information from different trojan detector instances and use the logical clock to determine session arrival order.

Typically, when ordering is required, an NF just needs to know the order in which *per flow state* is updated across instances, which is supported as above. However, our framework cannot ensure ordering between updates to non-perflow state by different instances.

## 6 Implementation

Our StreamNF prototype consists of an execution framework and datastore. The former is implemented atop Storm [1], a low latency stream processing system. StreamNF DAGs are translated into storm “topologies”.

Vertices are implemented as Storm “bolts” using our custom bolt interface which provides support for input message queues, client-side datastore library, statistics monitoring and logic for handling state during scaling, fault tolerance, and straggler mitigation. The root splitter is translated to a Storm “spout”; all other splitters execute as bolts. Scope-aware partitioning is implemented using the *CustomStreamGrouping* interface of Storm. Our prototype inherits the limitation of Storm that the logical topology cannot be changed at the runtime. Our datastore manager is based on Redis [6] - an in-memory key-value store. Redis provides intrinsic support for some of the operations in Table 1 (e.g., increment) and it allows implementation of custom store operations.

We reimplemented four NFs atop our framework:

**Network address translator (NAT):** Our NAT maintains the dynamic list of available ports in the datastore. When a new connection arrives, the NAT obtains an available port (by reading from the store). It then updates: 1) per-connection port mapping and, 2) network and transport layer packet counters for logging.

**Port scan detector:** We use [17] as our port scan detector. It detects scanning worms on a host by leveraging the fact that, for a benign host, the probability of successful connection attempts should be higher than that of unsuccessful ones. It keeps track of new connection initiation for each host and whether it was successful or not, and compares this against stats from other hosts. We implemented this NF using two logical vertices – firewall and analyzer – similar to Figure 6. The datastore manager notifies the firewall whenever the analyzer updates the decision to block a host.

**Trojan detector:** We implemented the trojan detector described in §5.4 using two logical vertices.

**Content caching proxy:** This NF caches all the HTTP replies. Cache replies are stored in the datastore and are shared between all instances. The NF also maintains the hits seen for each cache entry and an index to an already transferred portion of data.

The appendix discusses other StreamNF features.

## 7 Evaluation

We evaluate the performance and chain-wide guarantees offered by StreamNF. Specifically, we address these questions: (a) During normal operation, what overheads do state externalization, datastore manager, logical timestamps, and logging add? (b) When reallocations happen, what performance do StreamNF’s state management mechanisms offer relative to state-of-the-art? And, (c) How does StreamNF compare to state-of-the-art w.r.t., chain-wide/cross-NF properties (R4 and R5)? We evaluate the latter two questions in the context of

the StreamNF runtime management activities of elastic scaling, fault tolerance, and straggler mitigation. Our experiments use four Cloudlab [3] servers with two Intel Xeon E5-2630 v3 CPUs and a 10G NIC. The datastore and manager run on a dedicated server. The RTT between any two pairs of servers is  $60\mu\text{sec}$ . We collect a trace of packets on the link between our institution and a remote public cloud to use in our evaluation.

## 7.1 Overhead of building blocks

**State externalization:** To evaluate the overhead of externalizing the internal state, we consider five different models which reflect different optimizations: 1) All the state is local to NF. This represents a traditional NF and offers least overhead. 2) All state is externalized and non-blocking operations are used. 3) In addition, NFs cache per-flow and read-heavy state. 4) In addition, NFs do not wait for the ACK of a non-blocking operation. The framework is responsible for the retransmission.

In Figure 10, we present the per packet processing times<sup>9</sup>. Externalizing all the state operations increases the median packet processing time by  $130\mu\text{sec}$  and  $70\mu\text{sec}$  for the NAT and the proxy, resply., compared to traditional NFs. The network contributes most of this latency (e.g., NAT needs two RTTs: one for port mapping and another for logging stats). We don’t see a noticeable impact for scan and trojan detectors because they do not update the state on every packet. Caching improves median packet processing time by  $84\mu\text{sec}$  and  $41\mu\text{sec}$ , resply., for the NAT (needs 1 RTT to log stats at the store) and the proxy; scan and trojan detectors do not have any cache-able state. Not waiting for the ACK and offloading the retransmission of the operation to the framework further reduces the medium packet processing time to  $3.5\mu\text{sec}$  (NAT now needs 0 RTTs) and  $1.8\mu\text{sec}$ , resply., a *modest overhead*.

**Logical clock:** Recall that the splitter writes the logical clock value to the datastore to support fault tolerance. We see this adds a  $62\mu\text{sec}$  latency per packet (dominated by RTT). We optimize further by writing the clock to the store after every  $n^{\text{th}}$  packet.<sup>10</sup> The average overhead per packet reduces to  $6.37\mu\text{sec}$  and  $0.68\mu\text{sec}$  for  $n = 10, 100$ .

**Packet logging:** We evaluated two models of logging: 1) locally at the root splitter. 2) in the datastore. The former adds  $1\mu\text{sec}$ , whereas the latter adds  $71.5\mu\text{sec}$ .

**Datastore benchmarks:** We benchmarked the Redis

datastore using the workload imposed by our state operations. We found that for intrinsically-supported operations the store can offer  $\sim 120\text{K ops/s}$  (increment at  $129\text{K}$ , get at  $155\text{K}$ , set at  $121\text{K ops/s}$ ). Custom operations (conditional increment) can be supported at  $111\text{K ops/s}$ . The datastore can be easily scaled to support a greater rate of operations by simply adding multiple Redis nodes (ops/s scales linearly with nodes); crucially each state object is stored at exactly one node and hence no cross-node synchronization or coordination is needed.

## 7.2 State management

**Cross-instance state transfers:** We consider elastic scale up of our port scan detector. We replay our trace for 30s through a single instance; midway through our replay, we reallocate 1700 flows to a new instance, forcing a move of the state corresponding to these flows.

We compare StreamNF with OpenNF’s loss-free move operation; recall that StreamNF provides both loss-freeness and order preservation. StreamNF’s move operation takes 95% less time ( $0.311\text{ms}$  vs  $5.7\text{ms}$ ), because, unlike OpenNF, StreamNF does not need to transfer state. It notifies the datastore manager to update the task IDs associated with the state.

However, when tasks are caching the state, they are required to flush cached state (operations) before updating the task IDs. Even then, StreamNF is 86% better (because StreamNF only needs to flush *operations*).

**Cross-instance state sharing:** Continuing with the experiment above, we compare the performance of updates to shared state (per host counters at the firewall) across the two port scanner instances. To highlight StreamNF’s benefits, we impose a “toy” *strong* consistency requirement on this state, meaning that all updates are serialized according to some global order.

Figure 11 shows the per packet processing latency seen. StreamNF’s median latency is two orders of magnitude lower than OpenNF’s ( $6.8\mu\text{sec}$  vs  $.8\text{msec}$ ). For strong consistency, the OpenNF controller receives all packets from NFs; each is forwarded it to every instance; the next packet is released only after all instances ACK. StreamNF’s store simply serializes all instances’ ops.

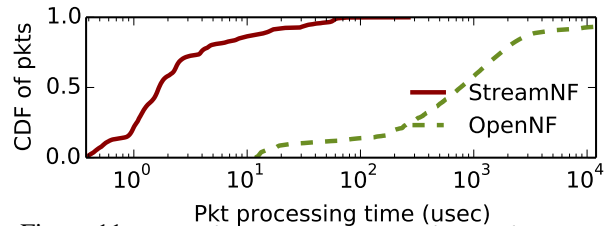


Figure 11: Per packet processing time with state sharing.

<sup>9</sup>The time between an NF receiving a packet and finishing processing.

<sup>10</sup>After a crash, this may lead the splitter to assign to a packet an already assigned clock value. To overcome this issue, the splitter starts with  $n + \text{last update}$ . This ensures that the clock values assigned to the packets represent their arrival order.

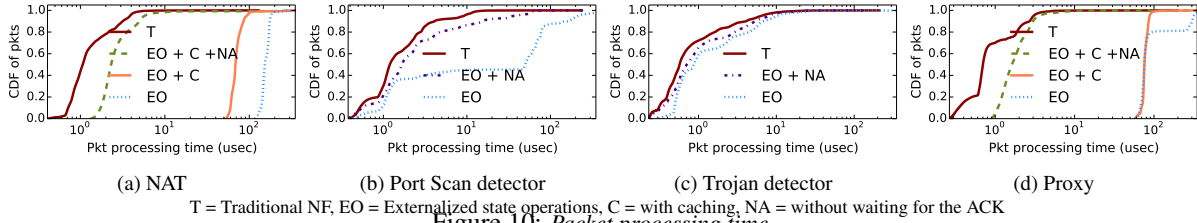


Figure 10: Packet processing time.

	25%load	50%load	75%load
Duplicate packets	9068	15346	24051
Duplicate state updates	187	429	661

Table 2: Duplicate packet and state update at the downstream port scan detector without duplicate suppression.

**State availability:** We compared StreamNF and FTMB’s state availability support. Similar to StreamNF, FTMB also logs each input packet. In addition FTMB also checkpoints NFs periodically. FTMB logs packets to a separate logger, and hence would face a similar overhead as StreamNF logging to the datastore. Thus, we evaluate the overhead imposed by checkpointing, and compare with StreamNF writing all state to a store. For this, we use a traditional NAT which does not have any state externalization. From Figure 12 (packets were arriving at 28k/s), we see that checkpointing in FTMB has a significant impact: the 99th%-ile latency is 34ms vs 13ms in StreamNF. Even the median latency is poor ( $2.3\mu s$  in StreamNF vs  $15.329\mu s$  in StreamNF); this is because checkpointing causes incoming packets to be buffered.

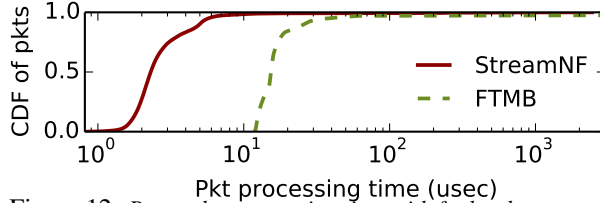


Figure 12: Per packet processing time with fault tolerance.

### 7.3 Chain-wide/Cross-NF properties

**Duplicate suppression:** Duplicate outputs need to be suppressed during straggler mitigation to ensure correct downstream operation. To measure the impact and amount of duplicate outputs, we emulated a straggler NAT by adding a random per packet delay between 3-10 $\mu s$ . A port scan detector is immediately downstream from the NAT. StreamNF launches a clone NAT instance according to §5.2. We vary the input traffic load. Table 2 shows the number of duplicate packets generated by the NAT instances under different loads, as well as the number of duplicate state updates (which happen whenever a duplicate packet triggers the scan detec-

tor to log a connection setup/teardown). Duplicates at the port scan detector generate spurious alerts. The numbers worsen at higher load. Existing frameworks cannot detect such duplicates due to upstream NF actions.

**Chain-wide ordering:** To evaluate the impact of chain-wide ordering, we consider a chain where there are 3 instances of an NF upstream from a single-instance Trojan detector. Each upstream instance either processes HTTP, SSH, or IRC traffic. To measure the accuracy of the trojan detector, we added the signature of a trojan at 11 different points in our traffic trace. We use three different workloads with varying processing speeds of the upstream NFs: W1) One of the upstream instances adds a random delay between 50-100 $\mu s$  to each packet. W2) Two of instances add the random delay. W3) All three add random delays. We observed that StreamNF’s use of chain-wide logical clocks helps the trojan detector identify all 11 signatures. We compare against OpenNF which does not offer any chain-wide guarantees; we find that OpenNF misses 7, 10, and 11 signatures across the W1-W3.

## 8 Conclusion

We presented a novel NFV framework called StreamNF. It provides a simple and flexible API to operators, hiding the low level details of managing NFV chains’ performance and correctness. We synthesize several ideas, some borrowed and others novel, to design intelligent state management external to NFs, minimize coordination among NF instances for shared state updates, and facilitate cross-NF ordering and duplicate suppression mechanisms. Together these ideas allow StreamNF to support high performance and fault tolerant NFV chains that satisfy key chain-wide requirements.

## References

- [1] Apache Storm. <http://storm.apache.org/>.
- [2] Cisco Network Service Header: draft-quinn-sfc-nsh-03.txt. <https://tools.ietf.org/html/draft-quinn-sfc-nsh-03>.
- [3] Cloud lab. <http://cloudlab.us/>.

- [4] Network functions virtualisation: Introductory white paper. [http://www.tid.es/es/Documents/NFV\\_White\\_PaperV2.pdf](http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf).
- [5] Nfv management and orchestration: An overview. <https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf>.
- [6] Redis: In-memory data structure store. <http://redis.io/>.
- [7] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 49–60, 2012.
- [8] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: Making the fast case common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, 1999.
- [9] M. Boucadair, C. Jacquenet, R. Parker, D. Lopez, P. Yegani, J. Guichard, and P. Quinn. Differentiated Network-Located Function Chaining Framework. Internet-Draft draft-boucadair-network-function-chaining-02, IETF Secretariat, July 2013.
- [10] A. Bremner-Barr, Y. Harchol, and D. Hay. Open-box: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 511–524. ACM, 2016.
- [11] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.
- [12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
- [13] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 543–546, 2014.
- [14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174. ACM, 2014.
- [15] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 49–54, 2015.
- [16] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: Simplifying middlebox modifications using stateanalyzr. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 239–253, 2016.
- [17] M.-S. Kim, H.-J. Kong, S.-C. Hong, S.-H. Chung, and J. W. Hong. A flow-based method for abnormal network traffic detection. In *Network operations and management symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 599–612. IEEE, 2004.
- [18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.
- [19] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.
- [20] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42. ACM, 2015.
- [21] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation*, NSDI'07. USENIX Association, 2007.
- [22] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM*

*SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38. ACM, 2013.

- [23] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 1. ACM, 2013.
- [24] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.
- [25] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1357–1369. ACM, 2015.
- [26] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, 2012.
- [27] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, pages 227–240. ACM, 2015.
- [28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

## Appendix: Addition StreamNF Features

**Non-deterministic values:** Non-deterministic values e.g. “gettimeofday” or “random” require special handling during fault tolerance and straggler mitigation to ensure output equivalence. Tasks are required to write every locally computed non-deterministic value in the datastore to provide determinism during fault tolerance. These values are used during the input replay phase to avoid divergence of internal state.

**Non-deterministic values in straggler mitigation:** To ensure that the state of a task and its clone do not diverge during straggler mitigation, the framework replaces the local computation of non-deterministic values with the datastore manager based computations. The datastore manager computes and stores each non-deterministic value. If a second request for a non-deterministic value comes with the same logical clock, it emulates the computation and returns the same value again.

**Asynchronous processing.** StreamNF offers synchronous and asynchronous processing. The NF specifies the mode of operation during initialization. In synchronous mode, the NF instance waits for the operation to execute and returns the result. In asynchronous mode, the instance registers a callback which is executed when the data is received from the datastore manager. Asynchronous mode enables the instance to process packets coming from other flows while waiting for the state update to execute.